A ROS-based System for an Autonomous Service Robot

Viktor Seib(✉), Raphael Memmesheimer, and Dietrich Paulus

Active Vision Group (AGAS), University of Koblenz-Landau,
Universitätsstr. 1, 56070 Koblenz, Germany
{vseib,raphael,paulus}@uni-koblenz.de
http://agas.uni-koblenz.de

Abstract. The Active Vision Group (AGAS) has gained plenty of experience in robotics over the past years. This contribution focuses on the area of service robotics. We present several important components that are crucial for a service robot system: mapping and navigation, object recognition, speech synthesis and speech recognition. A detailed tutorial on each of these packages is given in the presented chapter. All of the presented components are published on our ROS package repository: http://wiki.ros.org/agas-ros-pkg.

Keywords: Service Robots, SLAM, Navigation, Object Recognition, Human-Robot Interaction, Speech Recognition, Robot Face

1 Introduction

Since 2003 the Active Vision Group (AGAS) is developing robotic systems. The developed software was tested on different robots in the RoboCup[1] competitions. Our robot *Robbie* participated in the RoboCup Rescue league where it became world champion in autonomy (*Best in Class Autonomy Award*) in 2007 and 2008. It also won the *Best in Class Autonomy Award* in the RoboCup German Open in the years 2007 to 2011. Our robot *Lisa* depicted in Fig. 1 is the successor of Robbie and is participating in the RoboCup@Home league. Lisa's software is based on Robbie's, but was adopted and extended to the tasks of a service robot in domestic environments. Lisa has already won four times the 3rd place and in 2015 the 2nd place of the RoboCup German Open and is frequently participating in the finals of the RoboCup@Home World Championship. Further, in 2015 Lisa won the 1st place in the RoboCup@Home World Championship in Hefei, China. Additionally, Lisa has won the Innovation Award (2010) and the Technical Challenge Award (2012) of the RoboCup@Home league. Inspired by this success, we want to share our software with the ROS community and give a detailed instruction on how to use this software with other robots. Further information about our research group and robotic projects are available online[2] and information about Lisa at our project website[3].

[1] RoboCup website: http://www.robocup.org
[2] Active Vision Group: http://agas.uni-koblenz.de
[3] Team homer@UniKoblenz: http://homer.uni-koblenz.de

Fig. 1. The current setup of our robot Lisa. A laser range finder in the lower part is used for mapping and navigation. On top, an RGB-D camera, a high resolution camera and a microphone are mounted on a pan-tilt unit. Additionally, Lisa possesses an 6-DOF robotic arm for object manipulation and a screen displaying different facial expressions to interact with the user.

After a common environment setup, presented in Sec. 2, we will introduce the following ROS components in this chapter:

- First, we introduce a graphical user interface, `homer_gui` in Sec. 3.
- Second, we present nodes for mapping and navigation in Sec. 4.
- This will be followed by a description of our object recognition in Sec. 5.
- Finally, we demonstrate components for human robot interaction such as speech recognition, speech synthesis and a rendered robot face in Sec. 6

With the presented software, a ROS enabled robot will be able to autonomously create a map and navigate in a dynamic environment. Further, it will be able to recognize objects. Finally, the robot will be able to receive and react to speech commands and reply using natural language. The speech recognition is implemented by interfacing with an Android device that is connected to the robot. A robot face with different face expressions and synchronized lip movements can be presented on an onboard screen of the robot.

The presented software can be downloaded from our ROS software repository[4]. All example code in this chapter is also available online. Further, we provide accompanying videos that can be accessed via our YouTube channel[5].

[4] AGAS ROS packages: `http://wiki.ros.org/agas-ros-pkg`
[5] homerUniKoblenz on YouTube: `http://www.youtube.com/c/homerUniKoblenz`

2 ROS Environment Configuration

All contributed packages have been tested with Ubuntu 14.04 LTS (Trusty Tahr) and ROS Indigo. For convenience, the script `install_homer_dependencies.sh` (in the software repository) installs all needed dependencies. However, also the step-by-step instruction in each section can be used.

There are common steps to all presented packages that need to be taken to configure the ROS environment properly. To configure all paths correctly, it is necessary to define an environment variable `HOMER_DIR`. It should contain the path of the folder that contains the *catkin workspace* on your hard disk. For example, on our robot the path is `/home/homer/ros`. To set the environment variable `HOMER_DIR` to point to this path, type:

```
export HOMER_DIR=/home/homer/ros
```

Please replace the path by the one that you use. To test whether the variable was set correctly, type:

```
echo $HOMER_DIR
```

This command displays the path the variable points to.

Note that you need to set the path in every terminal. To avoid this, the path can be defined in your session configuration file `.bashrc` and will be automatically set for each terminal that is opened. In order to do this, please type:

```
echo "export HOMER_DIR=/home/homer/ros" >> ~/.bashrc
```

After setting the path, make sure that you did not change the directory structure after downloading the software from our repository. However, if you changed it, please adjust the paths written out after `$ENV{HOMER_DIR}` in the *CMakeLists.txt* file of the corresponding packages.

3 Graphical User Interface

Although a graphical user interface (GUI) is not necessary for a robot, it provides a lot of comfort in monitoring or controlling the state of the system. ROS already has `rviz`, a powerful tool for visualization of sensor data that can be extended with own plugins. Thus, we do not aim at creating our own visualization tool. Rather, in our package `homer_gui` we include `rviz`'s visualization as a plugin. This means that it is able to visualize everything that `rviz` does and can be adapted using the same configuration files. However, our GUI provides more than pure data visualization. It offers a convenient way of creating a map, defining navigation points and monitoring the navigation. Also, it enables the user to train and test new objects for object recognition in a convenient way and to monitor and define commands for human robot interaction. Additionally, basic state machine-like behavior and task execution can be defined in the GUI with only a few mouse clicks. The `homer_gui` is intended to run directly

on the on-board computer of the robot. However, it can also be used to control the robot's state and behavior remotely. Please note that the version of the `homer_gui` presented here is a reduced version, encompassing only functionality that we provide additional packages for. As we continue releasing more and more components from our framework, more tabs and widgets will be included in the `homer_gui`.

3.1 Background

The `homer_gui` package is a Qt-based application. It serves as the central command, monitoring and visualization interface for our robot. The main concept is to define several task specific tabs that provide a detailed control of all task specific components. As an example, in the currently provided `homer_gui` package we added the *Map* and *Object Recognition* tabs. These two tabs are closely linked to our mapping and navigation (see Sec. 4) and to our object recognition (see Sec. 5). Their functionality will be described in the corresponding sections, together with the related packages. The third tab in `homer_gui` is the main tab with the title *Robot Control*. Its purpose is the visualization of the sensor data and the robot's belief about the world. Further, some frequently used functionality of the robot can be accessed through buttons in this tab.

3.2 ROS Environment Configuration

Please make sure to take the configuration steps described in Sec. 2 before proceeding. In order to use the package `homer_gui` you have to install the following libraries:

- Qt: `http://qt.nokia.com`
- Rviz: `http://wiki.ros.org/rviz` (usually comes with ROS)
- PCL: `http://pointclouds.org/` (usually comes with ROS)

You can install the Qt libraries by opening a terminal and typing (all in one line):

```
sudo apt-get install libqt4-core libqt4-dev libqt4-gui
```

3.3 Quick Start and Example

To start the `homer_gui`, type

```
roslaunch homer_gui homer_gui.launch
```

A window, similar to Fig. 2 will appear. By itself, the GUI is not very exciting. Its full potential unfolds when used on a robot platform or e.g. with the packages described in Sec. 4 and Sec. 5. If started on a robot platform, the 3D view of the *Robot Control* tab will, among others, display the laser data and the robot model (Fig. 2).

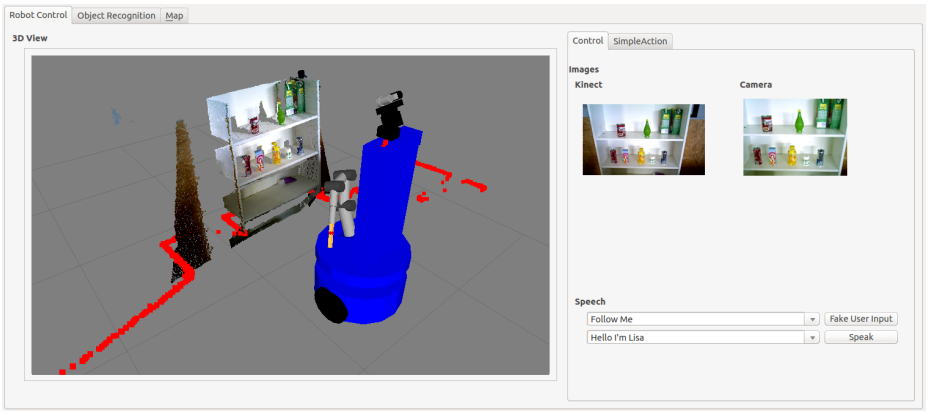


Fig. 2. Graphical user interface `homer_gui`. On the left the well known 3D view of `rviz` is included as plugin. On the right images of two connected cameras are displayed.

3.4 Package Description

Since the *Object Recognition* and *Map* tabs are described in detail in Sec. 4 and Sec. 5, this description will focus on the *Robot Control* tab. The main functionality of this tab is the 3D view on the left part of the tab. We included the `rviz` plugin into the 3D view widget. On startup, the `homer_gui` will load the file `homer_display.rviz` contained in the `config` folder of the package. Every `rviz` display configured in that file will also be available in our GUI. However, it can only be reconfigured using `rviz`.

The right side of the *Robot Control* tab contains two subtabs, *Control* and *SimpleAction*. The *Control* has two image widgets on top. The left one subscribes to the topic `/camera/rgb/image_raw`, the default image topic of the widespread RGB-D cameras (e.g. Kinect). The right one can be used to visualize other images as needed. It subscribes to the topic `/image` and in the same manner as the left widget, expects a message of type `sensor_msgs/Image`. The *Control* tab further contains two text line widgets with corresponding buttons. The first one is used to simulate recognized speech. When the button *Fake User Input* is pushed, the text in the corresponding text widget is published on the topic `/recognized_speech` as a `std_msgs/String` message. The second button is used to let the robot speak. Pushing the button *Speak* publishes a `std_msgs/String` message on the topic `/robot_face/speak`. If the `robot_face` described in Sec. 6 is running, the speech will be synthesized.

The second subtab, *SimpleAction*, is used to spontaneously create basic state machine-like behavior. So far, three simple actions can be used: *Drive to*, *Speak* and *Hear*. After selecting one of the actions on the left side of the tab, a parameter needs to be specified.

The *Speak* action takes a string that will be spoken by the robot (using the `robot_face` package described in Sec. 6) as parameter. The *Drive to* action

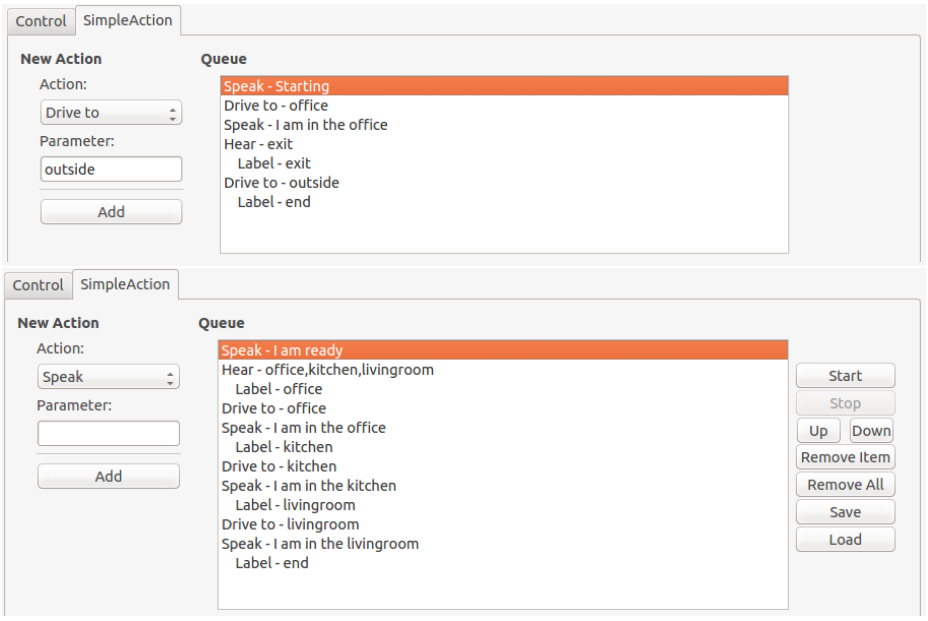


Fig. 3. Two examples of task sequences defined using the simple actions.

takes a point of interest (POI) name as string as its parameter. For now it is sufficiently to define a POI as a "named navigation goal". On execution, the robot will navigate to this POI in the map. For details on how to create POIs, please refer to Sec. 4. The *Hear* action is used for speech recognition and is the only action that takes multiple parameters, separated by commas (without spaces). If only one parameter is specified, the action will block until speech is recognized that matches the given parameter string. However, with multiple parameters a case distinction is achieved. Depending on the recognized words the robot can perform different behavior.

By using the *Add* button, an Action is added to the list of actions. Added actions, except *Hear*, can be moved inside the list. Clicking *Start* will start the execution until all actions are executed or the *Stop* button is pressed. Once a list of actions is defined it can be saved and loaded with the corresponding buttons.

Two examples for a command sequence defined using simple actions are presented in Fig. 3. The behavior of the robot defined in the upper image will be as follows: first, the robot will say "starting". After finishing the speech action, it navigates to the POI *office* and will say "I am in the office". After that the robot waits for the "exit" command. As soon as this command is received, the robot navigates to the POI *outside*.

The behavior defined in the lower image is an example for case distinction. The robot starts by saying "I am ready". After that it waits for one of three commands. Depending on whether it hears "office", "kitchen" or "living room,

it navigates to the corresponding POI and announces that it has reached its destination.

Thus, using the *Simple Action* tab basic robotic behavior can be achieved without programming. In the following sections further functionality of the `homer_gui` and the corresponding packages will be introduced.

4 Mapping and Navigation

The approach used in our software follows the algorithms described in [16] that were developed in our research group. To generate a map, the algorithm uses the robot's odometry data and the sensor readings of a laser range finder. The map building process uses a particle filter to match the current scan onto the occupancy grid. For navigation a path to the target is found using the A*-algorithm on the occupancy grid. The found path is post processed to allow for smooth trajectories. The robot then follows waypoints along the planned path.

The main benefit of our mapping and navigation compared to well established packages such as `hector_mapping`[6] and `amcl`[7] is its close integration with the graphical user interface `homer_gui`. It allows the user to define and manage different named navigation targets (points of interest). Further, maps can be edited by adding or removing obstacles in different map layers. Despite the points of interest and map layers, the grid maps created with our application are completely compatible with the ROS standard. The additional information is stored in a separate *yaml*-file that is only processed by the `homer_gui`, while the actual map data is not changed.

A video showing an example usage of the `homer_gui` with our mapping and navigation is available online[8].

4.1 Background

Mapping and Localization For autonomous task execution in domestic environments a robot needs to know the environment and its current position in that environment. This problem is called SLAM (Simultaneous Localization and Mapping) and is usually solved by a probabilistic Bayes formulation, in our case by a particle filter. Based on the position estimate from odometry the current laser scan is registered against the global occupancy grid. The occupancy grid stores the occupation probability for each cell and is used for path planning and navigation.

Navigation This approach is based on Zelinsky's path transform [17], [18]. A distance transform is combined with a weighted obstacle transform and used for navigation. The distance transform holds the distance per cell to a given

[6] Hector Mapping: `http://wiki.ros.org/hector_mapping`
[7] AMCL: `http://wiki.ros.org/amcl`
[8] Example video for mapping and navigation: `http://youtu.be/rH4pNq3nlds`

target. On the other hand, the obstacle transform holds the distance to the closest obstacle for each cell. This enables the calculation of short paths to target locations while at the same time maintaining a required safety distance to nearby obstacles.

Our navigation system merges the current laser range scan as a frontier into the occupancy map. A calculated path is then checked against obstacles in small intervals during navigation, which can be done at very little computational expense. If an object blocks the path for a given interval, the path is re-calculated. This approach allows the robot to efficiently navigate in highly dynamic environments.

4.2 ROS Environment Configuration

Please make sure to take the configuration steps described in Sec. 2 before proceeding. Further, the package `homer_gui` will be used in the examples presented here. Please also follow the configurations steps described in Sec. 3. The mapping and navigation packages require the *Eigen* library. To install this dependency, please type the following command in a terminal:

```
sudo apt-get install libeigen3-dev
```

4.3 Quick Start and Example

Before proceeding, do not forget to execute the `catkin_make` command in your ROS workspace.

Mapping To be able to use the mapping, your robot needs to publish odometry data on the topic `/odom` and laser range data on the topic `/scan`. Please refer to the ReadMe file[9] if your robot is publishing encoder ticks instead of odometry data. With the odometry and laser range data we are able to construct 2D occupancy grid maps using the `homer_mapping` package. However, the position of the laser range finder relative to the robot needs to be given. This is achieved by creating a transformation from `/base_link` to `/laser` for instance by adding:

```
<node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser_broadcaster"
args="0.0 0.0 0.1 0.0 0.0 0.0 1.0 /base_link /laser 100"/>
```

to your launch file. Please substitute the frame name of the laser range finder (here: `/laser`) by the frame `frame_id` of the `/scan` messages from your laser range finder. For further information on transformations and frames please check out the `tf` package documentation[10].

With these preparations completed the mapping can be started with the following command:

[9] A ReadMe is available online in the software repository.
[10] `tf` package documentation: `http://wiki.ros.org/tf`

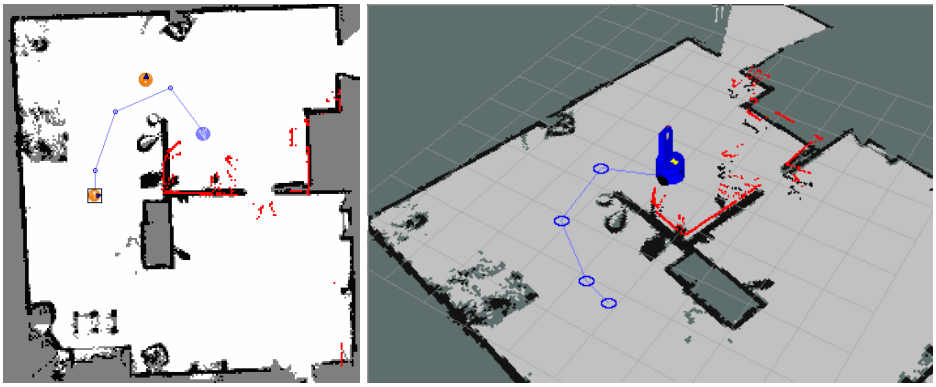


Fig. 4. 2D and 3D view of a map and a planned path (blue line). Red dots indicate the current laser scan, while orange circles in the 2D map represent named navigation targets (points of interest).

```
roslaunch homer_mapping homer_mapping.launch
```

The mapping node constructs the map incrementally and refines the current position of the robot using a particle filter SLAM approach. The map is published on the `/map` topic and the robot's pose on the topic `/pose`. The map can be inspected with `rviz` or our interface presented in Sec. 3 by typing

```
roslaunch homer_gui homer_gui.launch
```

and selecting the *Map* tab. An example visualization is shown in Fig. 4.

Navigation Prerequisite for navigation is the map and the robot's pose, both are computed by the `homer_mapping` package:

```
roslaunch homer_mapping homer_mapping.launch
```

For navigation, please additionally start the navigation package with the following command:

```
roslaunch homer_navigation homer_navigation.launch
```

The navigation node will publish velocity commands of type `geometry_msgs/Twist` on topic `/cmd_vel`. Please make sure that your robot subscribes to this topic.

If you are using `rviz` you can use the *2D Pose Estimate* button to localize the robot (if needed) and subsequently the *2D Nav Goal* button to define a target location for navigation and start navigating.

Navigation can also be started via command line. The equivalent command to the *2D Nav Goal* button is

```
rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped
```

After typing this command, press the *TAB*-button twice and fill in the desired target location. Alternatively, you can use the following command to start navigation

```
rostopic pub /homer_navigation/start_navigation
             homer_mapnav_msgs/StartNavigation
```

The latter option has additional parameters that allow you to define a distance to the target or to ignore the given orientation. In the next section, named target location, the so called points of interest (POIs), will be introduced. The following command can be used to navigate to a previously defined POI (in this example to the POI *kitchen*):

```
rostopic pub /homer_navigation/navigate_to_POI  homer_mapnav_msgs/
        NavigateToPOI
               "poi_name: 'kitchen'
               distance_to_target: 0.0
               skip_final_turn: false"
```

In all cases the navigation can be canceled by the command

```
rostopic pub /homer_navigation/stop_navigation std_msgs/Empty "{}"
```

As soon as the target location is reached, a message of type `std_msgs/Empty` is published on the topic `/homer_navigation/target_reached`. In case the navigation goal could not be reached, a message of type `homer_mapnav_msgs/TargetUnreachable` is published on the topic `/homer_navigation/target_unreachable` which holds the reason why the target could not be reached.

Please note that the full strength of our mapping and navigation packages comes from its close integration with the `homer_gui` package, which will be explained in the following section.

4.4 Using the `homer_gui` for Mapping and Navigation

This section describes the integration of the mapping and navigation with the `homer_gui` and the involved topics and messages. Please type the following commands to start all components needed for the examples in this section:

```
roslaunch homer_mapping homer_mapping.launch
roslaunch homer_navigation homer_navigation.launch
roslaunch homer_gui homer_gui.launch
```

A window showing the `homer_gui` should open. Please select the *Map* tab where you should see the current map and the robot's pose estimate similar to Fig. 5. Optionally, a previously created map can be loaded with the *Open...* button. After loading a map, usually the robot's pose needs to be changed. This is achieved by *Ctrl + LeftClick* in the map view. To change the robot's orientation please use *Ctrl + RightClick*, clicking on the point the robot should look at. This corresponds to the functionality encapsulated in the *2D Pose Estimate* button in

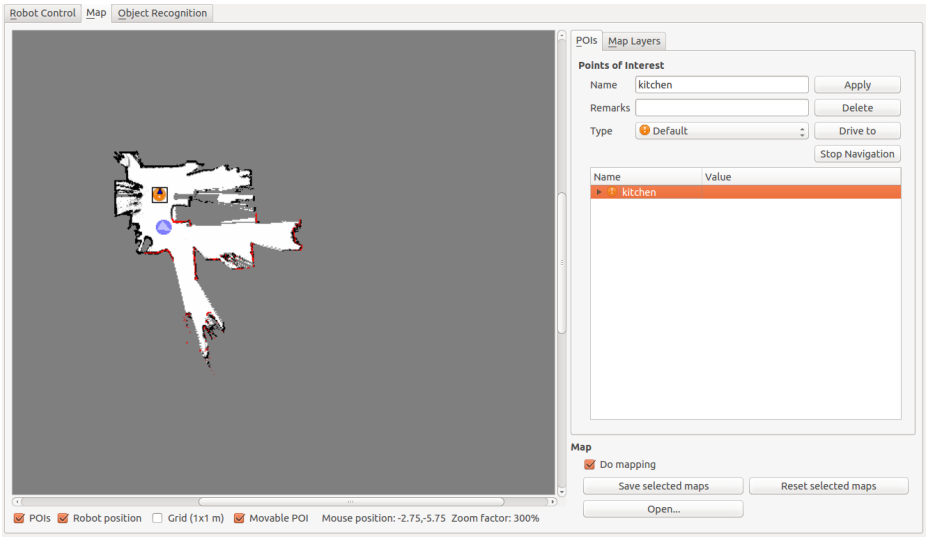


Fig. 5. The mapping tab in the `homer_gui`. On the left side the occupancy gridmap is shown. The current laser measurements are aligned to the map and drawn in red. The robot's pose estimate is denoted by the blue circle with an arrow. Orange circles are points of interest.

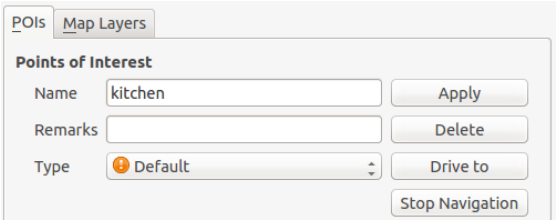


Fig. 6. The interface for adding and editing POIs, as well as to start and abort navigation.

`rviz`, however, by splitting the definition of the robot's location and orientation a more precise pose estimate can be given than if both have to be set at once.

However, the functionality of the *2D Nav Goal* button in `rviz` is handled completely differently in our system. In `rviz`, the *2D Nav Goal* button is used to define a navigation goal, a target orientation *and* start navigating towards this goal at once. In the `homer_gui` a click on the map adds a point of interest (POI) to the map view. After adding a POI symbol a name needs to be assigned on the right side of the tab (Fig. 5). A close-up of this dialog is shown in Fig. 6. A click on *Apply* informs the `homer_map_manager` that a new POI is available. The target orientation of a POI is denoted by a small arrow inside the POI symbol. To change its orientation a POI must be selected. A *Ctrl + LeftClick* on a location in

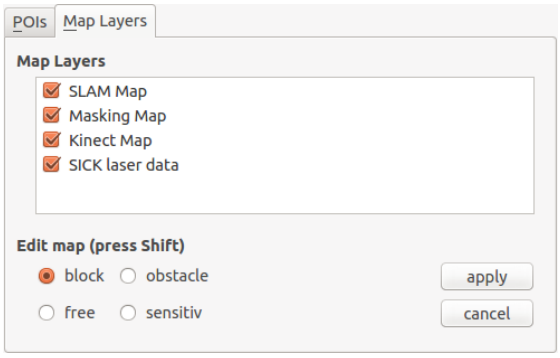


Fig. 7. The Map Layers interface to manually modify the existing map in different ways

the map defines the direction the robot should face after reaching the target. The specified orientation needs to be confirmed with the *Apply* button. An arbitrary number of POIs can be added to the map specifying all required locations for navigation. To actually navigate to a POI, it needs to be selected and the *Drive to* button pressed (Fig. 6). The *Stop navigation* button allows to abort navigation and stop the robot. Please note that the *Drive to* button uses the same topic and message as the *2D Nav Goal* button in `rviz`. However, the location in `rviz` is determined from a click on the map whereas here the location is taken from the predefined POI.

The `homer_gui` further allows to use different map layers. This is used e.g. if the robot is equipped with only one laser range finder on a fixed height that is not capable of detecting all obstacles in the area. The map layers are shown in the *Map Layers* subtab of the *Map* tab (Fig. 7). The *SLAM Map* layer is the normal occupancy grid for navigation. The *laser data* layer only shows the current laser scan, while the *Kinect Map* layer is currently not used, but will contain sensor readings from an RGB-D camera in the future. All selected layers are merged into the *SLAM Map* and used for navigation and obstacle avoidance. The *Masking Map* layer is used to add additional obstacles that the robot was not able to perceive automatically. To modify the *Masking Map*, please select it and press *Shift + LeftClick* into the map view. A red square will appear whereby its corners can be dragged to give it the desired shape and location. A click on *apply* will finally add the modification to the layer. There are different types of modifications that can be added:

block : An area that the robot will consider as an obstacle that will be avoided in any case (e.g. stairs or other not perceived obstacles).

obstacle : Similar to *block*, however this type can be used to allow manipulation in this area while navigation will still be prohibited.

free : This area will be considered as free regardless of the sensor readings or the stored map data. This is useful if navigating with a fixed map where an object has moved or a closed door is now open.

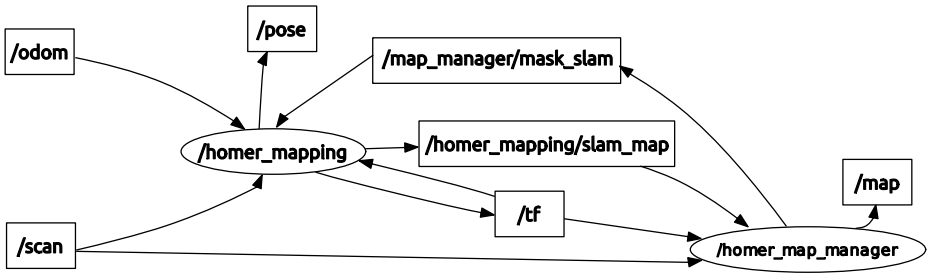


Fig. 8. ROS graph of the nodes inside the `homer_mapping` package with the most important topics.

sensitive : This type enables you to define high sensitive areas where obstacles are faster included into the map. This is useful for known dynamic parts of the map, for instance door areas.

These modifications can be added to any map layer. However, we advise to only modify the *Masking Map* manually since other layers are continuously updated by sensor readings and thus might alter the manual modifications.

4.5 Package Description and Code Examples

The graph in Fig. 8 shows the two nodes that are encapsulated in the `homer_mapping` package with the most important topics. The `homer_mapping` node is in charge of the mapping and the associated algorithms. The node `homer_map_manager` handles the loading and storage of maps, as well as manages the POIs and map layers. The most important topics are explained in the following:

Mapping Publishers

`/pose` (`geometry_msgs/PoseStamped`): The current pose estimate in the `/map` frame as calculated by the particle filter.

`/homer_mapping/slam_map` (`nav_msgs/OccupancyGrid`): The current map as output from the SLAM algorithm (without any map layers). The default publishing frequency is 2 Hz.

`/map` (`nav_msgs/OccupancyGrid`): The current map enhanced with map layers.

Mapping Subscribers

`/odom` (`nav_msgs/Odometry`): Current robot odometry values, needed by the particle filter for the SLAM algorithm.

`/scan` (`sensor_msgs/LaserScan`): Current laser measurements, needed by the particle filter for the SLAM algorithm.

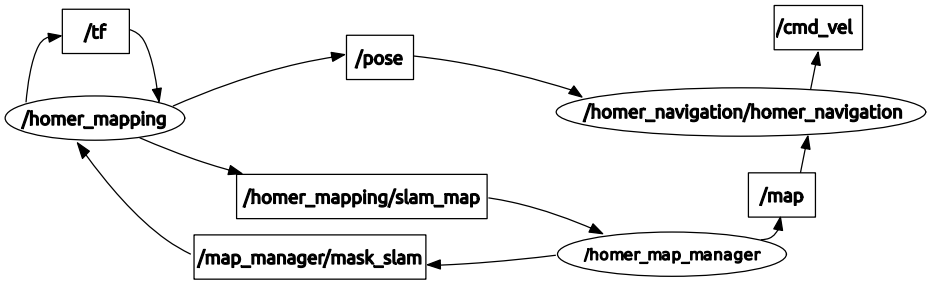


Fig. 9. ROS graph of the nodes inside the `homer_navigation` package with the most important topics.

Mapping Services

`/homer_map_manager/save_map` (`homer_mapnav_msgs/SaveMap`): Saves the current occupancy map with all POIs and layers.
`/homer_map_manager/load_map` (`homer_mapnav_msgs/LoadMap`): Loads a map from a specified file path.

The graph in Fig. 9 shows the `homer_navigation` node with its most important topics. As the navigation relies on the node `homer_mapping` and `homer_map_manager`, these nodes need to be running for the robot to be able to navigate. The `homer_navigation` node handles the path planning algorithm and sends control commands to the robot. The most important topics are explained in the following. For a better overview not all of these topics are shown in Fig. 9.

Navigation Publishers

`/cmd_vel` (`geometry_msgs/Twist`): Velocity commands calculated by the navigation node are published on this topic.
`/homer_navigation/target_reached` (`std_msgs/Empty`): When the robot reaches a navigation goal a message will be send on this topic.
`/homer_navigation/target_unreachable` (`homer_mapnav_msgs/TargetUnreachable`): The system is notified on this topic if a target is not reachable and the navigation is canceled.

Navigation Subscribers

`/map` (`nav_msgs/OccupancyGrid`): The current map with all defined map layers merged is used for path calculation and obstacle avoidance.
`/pose` (`geometry_msgs/PoseStamped`): The current pose of the robot for path planning.
`/scan` (`nav_msgs/LaserScan`): The current laser scan is used for obstacle avoidance.
`/homer_navigation/start_navigation` (`homer_mapnav_msgs/StartNavigation`): Plans a path and starts navigating to a target location.

`/homer_navigation/navigate_to_POI` (`homer_mapnav_msgs/NavigateToPOI`):
Plans a path and starts navigating to a given POI.

`/homer_navigation/stop_navigation` (`homer_mapnav_msgs/StopNavigation`):
Stops the current navigation task.

To use the mapping and navigation in your own application, be sure to include the `homer_mapnav_msgs` as *build* and *run* dependency in your package's `package.xml` file. Further, `homer_mapnav_msgs` has to be added to the `find_package` section and as a dependency to the `catkin_package` section in the `CMakeLists.txt` file. As with the command line examples from the *Quick Start* section we provide an example for navigating to a location and to a predefined POI by its name. An example header file for navigation could look like this:

```
#include <ros/ros.h>
#include <tf/tf.h>
#include <homer_mapnav_msgs/NavigateToPOI.h>
#include <homer_mapnav_msgs/StartNavigation.h>

class NavigationExample {
  public:
    NavigationExample(ros::NodeHandle nh);
    virtual ~NavigationExample(){};

    void driveToPOI(std::string name, float distance_to_target = 0.03);
    void driveToPosition(float x, float y, float z,
              float orientation_in_rad = 0.0,
              float distance_to_target = 0.03)
  private:
    void targetReachedCallback(
            const std_msgs::Empty::ConstPtr& msg);
    void targetUnreachableCallback(
            const homer_mapnav_msgs::TargetUnreachable::ConstPtr& msg);
    ros::Publisher navigate_to_poi_pub_;
    ros::Publisher start_navigation_pub_;
    ros::Subscriber target_reached_sub_;
    ros::Subscriber target_unreachable_sub_;
};
```

In your class implementation you can use the following code snippets. First, include the example header and initialize all subscribers and publishers in the constructor:

```
#include "NavigationExample.h"

NavigationExample::NavigationExample(ros::NodeHandle nh)
{
  navigate_to_poi_pub_ = nh->advertise<homer_mapnav_msgs::NavigateToPOI>
      ("/homer_navigation/navigate_to_POI", 1);
```

```
  start_navigation_pub_ = nh->advertise<homer_mapnav_msgs::
      StartNavigation>("/homer_navigation/start_navigation", 1);

  target_reached_sub_ = nh->subscribe<std_msgs::Empty>
      ("/homer_navigation/target_reached", 1,
      &NavigationExample::targetReachedCallback, this);

  target_unreachable_sub_ = nh->subscribe<homer_mapnav_msgs::
      TargetUnreachable>("/homer_navigation/target_unreachable", 1,
      &NavigationExample::targetUnreachableCallback, this);
}
```

The following function is used to navigate to a predefined POI.

```
void NavigationExample::driveToPOI(std::string name,
      float distance_to_target)
{
    ROS_INFO_STREAM("DRIVING TO "+ name);
    homer_mapnav_msgs::NavigateToPOI msg;
    msg.poi_name = name;
    msg.distance_to_target = distance_to_target;
    navigate_to_poi_pub_.publish(msg);
}
```

Further, the following function allows to navigate to arbitrary coordinates in the `/map` frame.

```
void NavigationExample::driveToPosition(float x, float y, float z,
                      float orientation_in_rad,
                      float distance_to_target)
{
  homer_mapnav_msgs::StartNavigation start_msg;
  start_msg.goal.position.x = x;
  start_msg.goal.position.y = y;
  start_msg.goal.position.z = z;
  start_msg.goal.orientation =
  tf::createQuaternionMsgFromYaw(orientation_in_rad);
  start_msg.distance_to_target = distance_to_target;
  m_start_navigation_pub.publish(start_msg);
}
```

Finally, the following two callbacks provide a feedback on whether the target location could be reached or not:

```
void NavigationExample::targetReachedCallback(
      const std_msgs::Empty::ConstPtr& msg)
{
```

```
    ROS_INFO_STREAM("Reached the goal location");
}
void NavigationExample::targetUnreachableCallback(
      const homer_mapnav_msgs::TargetUnreachable::ConstPtr& msg)
{
    ROS_WARN_STREAM("Target unreachable");
}
```

5 Object Recognition

The ability to recognize objects is crucial for robots and their perception of the environment. Our object recognition algorithm is based on SURF feature clustering in Hough-space. This approach is described in detail in [14]. We applied it in the Technical Challenge of the RoboCup 2012 where we won the 1st place. Additionally, with an extended version of the algorithm we won the 2nd place in the Object Perception Challenge of the RoCKIn competition in 2014.

A video showing an example usage of the `homer_gui` with our object recognition is available online[11].

5.1 Background

Our object recognition approach is based on 2D camera images and SURF features [2]. The image processing pipeline for the training procedure is shown in Fig. 10. In order to train the object recognition classifier an image of the background and of the object has to be captured. These two images are used to calculate a difference image to segment the object. From segmented object we extract a number of SURF features f. A feature is a tuple $f = (x, y, \sigma, \theta, \delta)$ containing the position (x, y), scale σ and orientation θ of the feature in the image, as well as a descriptor δ. Thus, the features are invariant towards scaling, position in the image and in-plane rotations. Further images with a different object view need to be acquired and added to the object model in the database to capture the object's appearance from all sides.

The image processing pipeline for the recognition step is shown in Fig. 11. Since no information about the background is available during the object recognition phase, SURF features are extracted from the whole input image. The

[11] Example video for object recognition: `https://youtu.be/dptgFpu7doI`

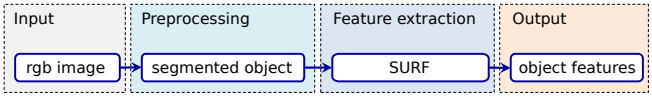


Fig. 10. Image processing pipeline for the training phase.

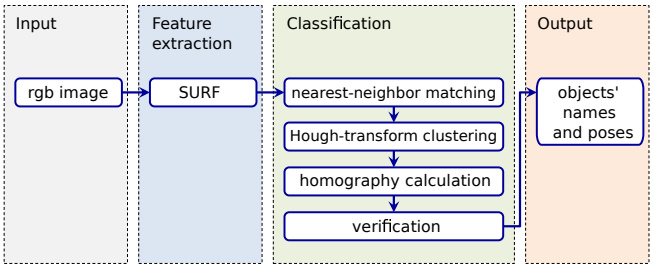


Fig. 11. Image processing pipeline for the recognition phase.

obtained features are then matched against the features stored in the object database. For fast nearest-neighbor and distance computation in the high dimensional descriptor space we use an approximate nearest neighbor approach [10]. Since simple distance thresholds do not perform well in high dimensional space, Lowe introduced the distance ratio [7], which is used here to sort out ambiguous matches. The result of feature matching is a set of matches between features extracted from training images and the scene image. This set may still contain outliers, i. e. matches between learned features and the scene's background.

The feature matches are clustered in a four dimensional Hough-space representing the 2D position of the object's centroid in the image (x, y), it's scale σ and it's rotation θ. The goal is to find a consistent object pose in order to eliminate false feature correspondences. Each feature correspondence is a hypothesis for an object pose and is added to the corresponding bin in the accumulator. As suggested in [7] and [5], to reduce discretization errors, each hypothesis is added to the two closest bins in each dimension, thus resulting in 16 accumulator entries per feature correspondence. As a result of the Hough-transform clustering all features with consistent poses are sorted into bins, while most outliers are removed since they do not form maxima in Hough-space.

In the next step, bins representing maxima in Hough-space are inspected. A perspective transformation is calculated between the features of a bin and the corresponding points in the database under the assumption that all features lie on a 2D plane. As most outliers were removed by discarding minima in Hough-space, a consistent transformation is obtained here. Random Sample Consensus (RANSAC) is used to identify the best homography for the set of correspondences. The homography with most point correspondences is considered to be the correct object pose. Finally, the object presence in the scene is verified by comparing the number of matched features of the object with the number of all features in the object area. For further details on the presented algorithm, please refer to [14].

We applied the described algorithm in the Technical Challenge of the RoboCup@Home World Championship 2012 in Mexico-City where we won the 1st place. The cropped input image as well as the recognition result of the Technical Challenge are shown in Fig. 12.

Fig. 12. The input image for object recognition as acquired by our robot during the Technical Challenge of the RoboCup (a) and the output image depicting the recognition results (b). During training and recognition an image resolution of 8 MP was used.

5.2 ROS Environment Configuration

Please make sure to take the configuration steps described in Sec. 2 before proceeding. The following *Quick Start and Example* section will use the `homer_gui` package described in Sec 3. Please make sure to follow the steps described there for the *ROS Environment Configuration*, as well.

The nodes for object recognition are contained in the `or_nodes` package. This package depends on the following packages that are also provided in our software repository:

- `or_libs` This package contains all algorithms used for object recognition.
- `or_msgs` All messages used by the `or_nodes` package are contained here.
- `robbie_architecture` Core libraries from our framework that other packages depend on are enclosed here.

All libraries needed to run these packages are installed with ROS automatically. However, you will need a camera to follow the presented examples. Although our approach uses RGB data only, in our experience the most widespread and

ROS supported camera is a Kinect-like RGB-D camera. For the camera in our example we use OpenNI that you can install by typing (all in one line):

```
sudo apt-get install ros-indigo-openni-camera
 ros-indigo-openni-launch
```

You can use any other camera, of course.

5.3 Quick Start and Example

Before proceeding, please make sure that you completed the *ROS Environment Configuration* for the object recognition packages. Further, do not forget to execute the `catkin_make` command in your ROS workspace.

To recognize objects you need to connect a camera to your computer. In our example we use a Microsoft Kinect. After connecting the device, start the camera driver by typing

```
roslaunch openni_launch openni.launch
```

This should advertise several topics, including `/camera/rgb/image_rect_color`. This topic will be assumed as the default topic that our application uses to receive camera images from. You can specify a different topic by changing the parameter `/OrNodes/sInputImageTopic` inside the `params_kinect.yaml` file. This file is located in the `config` folder inside the `or_nodes` package.

You can start the object recognition by typing the following command:

```
roslaunch or_nodes or_nodes.launch
```

This will start the necessary nodes, as well as load the default object `lisa`. Please start now the `homer_gui` described in Sec. 3 by typing

```
roslaunch homer_gui homer_gui.launch
```

As soon as the main window appears, select the *Object Recognition* tab on the upper part of the window. In the tab you are now in please again select the *Object Recognition* on the right part of the screen. If everything works as expected, you should see the camera image of the Kinect in the top left image widget of the *Object Recognition* tab (see Fig. 13). Since a default object is already loaded, we can immediately test the recognition. In order to do this, click on the *Start Recognition Loop* button in the lower right part of the window. Now the object recognition is performed on a loop using the incoming camera images of the Kinect. Please go back to page 2 where an image of our robot Lisa is shown. If you point the kinect at that image, you should see a recognition result similar to Fig. 14. As soon as you are done testing, press the *Stop Recognition Loop* button. Please note that for obvious reasons we could not test this example with the final version of the book and used a printed image of the robot instead.

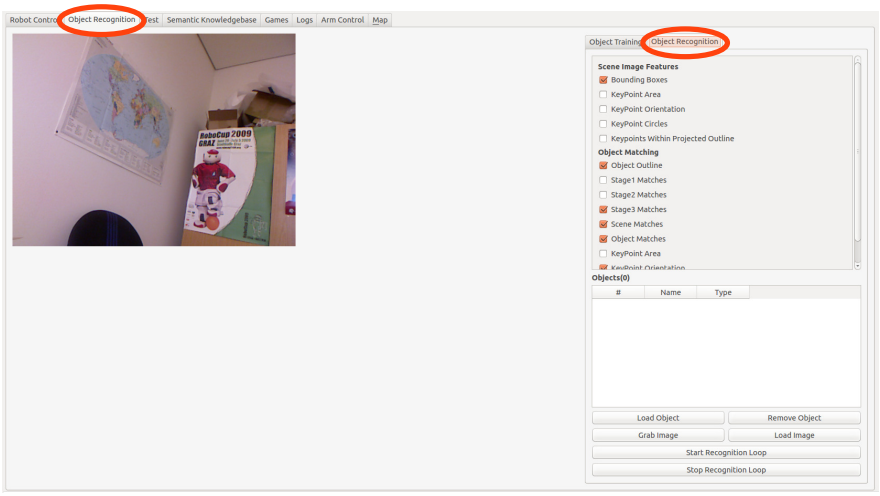


Fig. 13. Graphical user interface `homer_gui` with the current image of the Kinect in the upper left image widget. The red ellipses indicate the selected *Object Recognition* tabs.

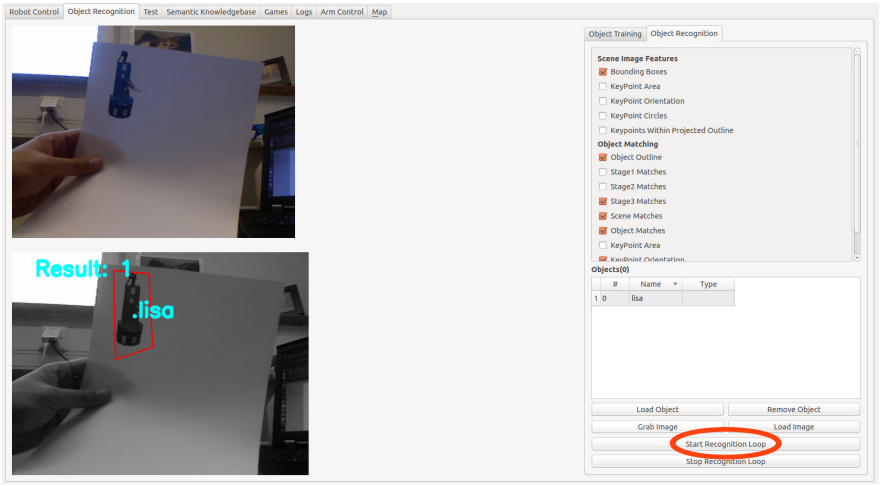


Fig. 14. Graphical user interface `homer_gui` displaying the recognition result of our robot Lisa in the bottom left image widget. The red ellipse indicates the button to start the recognition.

5.4 Using the `homer_gui` for Object Learning and Recognition

This section describes the integration of the object recognition and training pipelines with the `homer_gui`. Please connect an RGB-D camera (or any camera

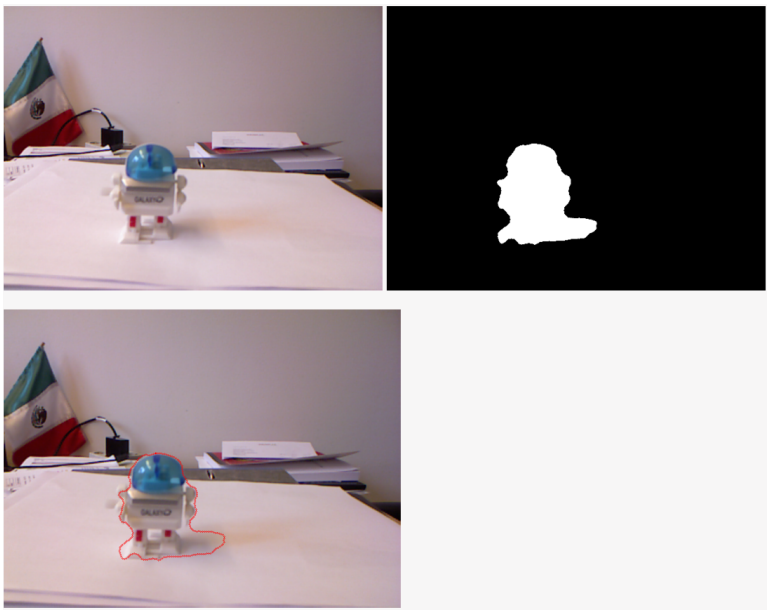


Fig. 15. The image widget part of the *Object Recognition* tab is shown. The top left image shows the current camera image. The top right image shows the object mask obtained by subtracting the background and foreground images. The bottom image shows the outline of the object according to the mask. Only features inside this outline will be saved for this object view.

you like) with your computer and type the following commands to start all nodes that are used in the following use case

```
roslaunch or_nodes or_nodes.launch
roslaunch homer_gui homer_gui.launch
roslaunch openni_launch openni.launch
```

Replace the last command by the launch-file of your camera driver.

To train a new object, click on *Object Recognition* tab in the `homer_gui` and in the appearing window on the tab *Object Training*. As in the quick start example, you should see the current camera image in the top left image widget. To obtain better results, point the camera at a static background then click *Grab Background*. The next incoming image on the image source topic (default: `/camera/rgb/image_rect_color`) will be processed as background image. The background image will appear in the bottom left widget (received over the topic `/or/obj_learn_primary`) and a grayscale version of the image will appear in the top right widget (received over the topic `/or/debug_image_gray`). Now place an object that you want to train in front of the camera and click on *Grab Foreground*. Again, the next incoming image on the image source topic will be processed. The result is sent to the `homer_gui` and displayed. The bottom left widget shows the

outline of the object, while the top right widget displays the extracted object mask (Fig. 15).

In some cases, the initial object mask is far from being optimal. Therefore, several thresholds need to be adjusted in the GUI using the three sliders and one checkbox on the right side. Each change will be immediately displayed in the colored image with the outline and the mask image. Note that the initially displayed state of the sliders and the checkbox must not necessarily reflect their real states as the GUI might have been started before the object learning node. It is therefore advised to change each of the sliders and re-activate the checkbox and see whether the segmentation improves. The semantics of the sliders and the checkbox are as follows. If the checkbox is checked, only the largest separated segment of the difference between the background and the foreground will be used. The threshold determining the separation between the foreground and background is set with the *Background Deletion Threshold* slider. Around the segmented mask, a morphological opening is performed. The radius for this operation is determined with the *Mask Open Radius* slider. Finally, an additional border can be added to the mask by adjusting the *Additional Border* slider. The obtained mask might contain a small portion of the table that was segmented due to the shadow of the object. Usually, this has no negative effect on the recognition result, as long as the table plane is homogeneous without many features.

To save the obtained object view, click on the *AddImage* button. Optionally, you can specify a name for the image in the text field next to the button. An image can be removed again by selecting the image in the list on the right and clicking the button *RemoveImage*. By clicking the *Reset* button, all images saved so far will be removed. Several views of the object (about 8 to 12) should be acquired and saved by repeating the described procedure. After obtaining enough object views, the object file has to be saved by specifying a name and clicking the *Save* button (top right corner of the window). The resulting object file will be the specified object name with the file extension *.objprop*. The file will be placed in the package `or_nodes` inside the folder `objectProperties`.

Besides learning new objects by the described procedure, the *Object Training* tab offers some more functionalities. Instead of grabbing the background and foreground images, images can also be loaded from the hard disk using the buttons *Load Background* and *Load Foreground*, respectively. Further, with the *Load Object* button, a previously saved object file can be loaded and more images added or removed.

To recognize a learned object, select the *Object Recognition* tab on the right. Please note that the `homer_gui` is still in development and the numerous checkboxes in upper right part the *Object Recognition* subtab have no functionality so far. It is planned to use these checkboxes in future to display additional debug information in the recognition result image widget. Please click on the *Load Object* button to select an object for recognition. Note that only loaded objects will be used for recognition. To remove a loaded object, the corresponding object has to be selected and subsequently the *Remove Object* button pushed.

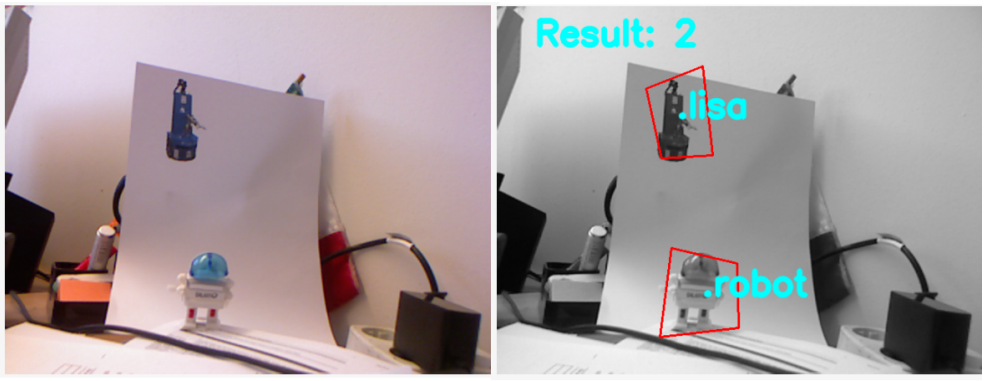


Fig. 16. Recognition example of the default object `lisa` and the learned object `robot`. The input image is shown on the left, the recognition results and bounding boxes are shown on the right.

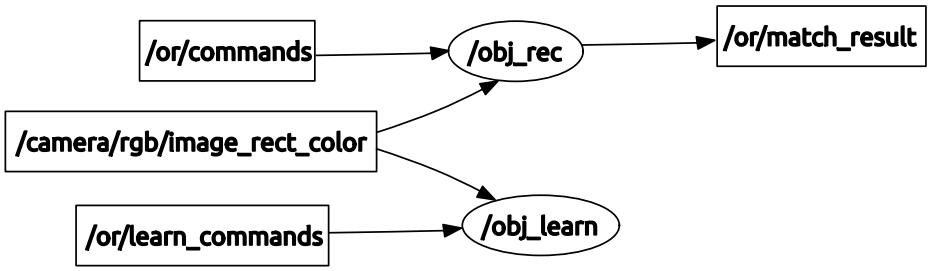


Fig. 17. ROS graph of the nodes inside the `or_nodes` package with the most important topics.

With the desired objects loaded, the recognition can be started by pressing the *Grab Single Image*, the *Load Image* or the *Start Recognition Loop* buttons. While the *Load Image* button is used to recognize objects on an image from the hard disk, the other two buttons use the incoming images from the camera sensor. In case the recognition loop was started, recognition runs continuously on the incoming image stream and can be stopped by a click on the *Stop Recognition Loop* button. The recognition result is sent as a `or_msgs/OrMatchResult` message on the topic `or/match_result`. Additionally, a `std_msgs/Image` message is published on the topic `/or/obj_learn_primary` displaying the recognition results (Fig. 16).

5.5 Package Description and Code Examples

The graph in Fig. 17 shows the two nodes contained in the `or_nodes` package with the most important topics. The `obj_learn` node is used for object learning

and the node `obj_rec` for object recognition. The most important topics are explained in the following:

Publishers

`/or/match_result` (or_msgs/OrMatchResult): Recognition results are published on this topic.
`/or/debug_image_gray` (std_msgs/Image): Debug images and extracted object mask images are published on this topic.
`/or/obj_learn_primary_color` (std_msgs/Image): Topic with images showing the object outline during learning and the recognition results.

Subscribers

`/camera/rgb/image_rect_color` (std_msgs/Image): Input image topic for object recognition and learning.
`/or/learn_commands` (or_msgs/OrLearnCommand): Topic for commands regarding the object learning procedure.
`/or/commands` (or_msgs/OrCommand): Topic for commands regarding object recognition.

To use the object recognition in your own application, be sure to include the `or_msgs` as *build* and *run* dependency in your package's `package.xml` file. Further, `or_msgs` has to be added to the `find_package` section and as a dependency to the `catkin_package` section in the `CMakeLists.txt` file. The easiest way to load objects for recognition is to include the following line in your launch file:

```
<rosparam param="or_objects">lisa,robot</rosparam>
```

This line sets the parameter `or_objects` on the ROS parameter server and assigns it the object names. The object names must be separated by a comma without spaces. In this example the object files `lisa.objprop` and `robot.objprop` will be loaded automatically on startup. The object files have to be located inside the *objectProperties* folder in the `or_nodes` package. Note that this parameter has to be set before launching the object recognition launch file. In case this parameter is not set, default objects from the legacy (deprecated) configuration file are loaded (see below). An example header for object recognition could look like this:

```
1  #include <ros/ros.h>
2  #include <or_msgs/OrCommand.h>
3  #include <or_nodes/src/Modules/ORControlModule.h>
4  #include <or_msgs/OrMatchResult.h>
5  #include <or_msgs/MatchResult.h>
6  #include <or_msgs/RecognizeImage.h>
7
8  class ObjectsExample {
9    public:
10     ObjectsExample(ros::NodeHandle nh);
```

```
    virtual ~ObjectsExample(){};
  private:
    void recognizeObjects();
    void orResultsCallback(const or_msgs::OrMatchResult::ConstPtr& msg);
    void recognizeImagePart(const sensor_msgs/Image &img_msg);
    ros::Publisher or_command_pub_;
    ros::Subscriber or_result_sub_;
    ros::ServiceClient or_client_;
};
```

In your class implementation you can use the following code snippets. First, include the example header and initialize all subscribers and publishers in the constructor:

```
#include "ObjectsExample.h"

ObjectsExample::ObjectsExample(ros::NodeHandle nh) {
    or_command_pub_ = nh.advertise<or_msgs::OrCommand>
        ("/or/commands", 10);
    or_result_sub_ = nh.subscribe("/or/match_result", 1,
        &ObjectsExample::or_result_callback, this );
    or_client_ = nh.serviceClient<or_msgs::RecognizeImage>
        ("/or/recognize_object_image");
}
```

The following function calls the recognition on the input image topic:

```
void ObjectExample::recognizeObjects()
{
    or_msgs::OrCommand msg;
    msg.command = ORControlModule::GrabSingleImage;
    or_command_pub_.publish(msg);
}
```

The recognition result is obtained in this callback:

```
void ObjectExample::orResultsCallback(const
     or_msgs::OrMatchResult::ConstPtr& msg)
{
  for(MatchResult mr : msg->match_results)
  {
    ROS_INFO_STREAM("Recognized object " << mr.object_name);
    // mr.b_box is the bounding box of this object
  }
}
```

In some application you might not want to use the whole input image for recognition, but only a certain region of interest. This is especially the case when you have segmented an object using geometrical information. The extracted RGB

image part of the segmented object can then be put into an `sensor_msgs/Image` message and a service call used for recognition. The following code snippet shows an example:

```
void ObjectExample::recognizeImagePart(const sensor_msgs/Image &img_msg)
{
  or_msgs::RecognizeImage srv;
  srv.request.image = img_msg;
  if(or_client_.call(srv))
  {
    // this vector contains all recognized objects inside img_msg
    std::vector<std::string> result = srv.response.names;
  }
  else ROS_ERROR_STREAM("Failed to call service!");
}
```

Finally, a word about configuration files of the object recognition. These packages contain a mixture of `.xml` and `.yaml` files for configuration inside the `config` folder of the `or_nodes` package. The `.xml` files are the deprecated configuration files from the pre-ROS era of our software framework. On the other hand, the `.yaml` files are the well known configuration files of ROS. The most important files are the `params_asus.yaml` and `params_kinect.yaml`. Both define same parameters, but differ in the default topic for the image input. Since the Kinect sensor was used in the examples above, we continue with it. The default content of the `params_kinect.yaml` is:

```
/OrNodes/sConfigFile:          /config/custom.xml
/OrNodes/sProfile:             drinks
/OrNodes/sInputImageTopic:     /camera/rgb/image_rect_color
/OrNodes/debugOutput:          /true
/OrNodes/debugOutputPath:      /default
```

The first value is the *package relative* path to the pre-ROS custom configuration file and the second line defines the profile to be used inside the custom configuration file. There are two `.xml` files: `default.xml` and `custom.xml`. The file `default.xml` is automatically loaded on startup. It contains all parameters necessary for object learning and object recognition, encapsulated in the profile `default`. Normally, you don't need to change values in this file. To specify custom values, please use the file `custom.xml` that is the default custom file in the `.yaml` file. Currently, the profile `drinks` is defined in `custom.xml` that overrides the default value of `ObjectRecognition/sLoadObject`. This is the parameter specifying the object files that should be loaded on startup in case the parameter `or_objects` is missing on the ROS parameter server. When loading the configuration files, a merged.xml is generated by the config loader that contains all available profiles from `default.xml` and `custom.xml`. The third line in the `.yaml` file is the default image topic that the object recognition is using. The last two lines define the debug output. However, they are not fully integrated, yet.

6 Human Robot Interaction

Human-Robot interaction is another basic component needed by service robots. We provide a ROS package coupled with an Android app for speech recognition and synthesis. It integrates with any modern Android device and enables the robot to use the speech recognition and synthesis of the Android system. Android provides speech recognition and synthesis for many languages and also works completely offline, once the dictionaries have been downloaded.

As a second component of the presented interaction system we present our robot face as introduced and evaluated in [13]. Apart from a neutral face expression it can show 6 other emotions. The robot face captures the speech output of the robot and moves the lips synchronously to the spoken words.

A video showing an example animated face with synthesized speech is available online[12].

6.1 Background

Speech Recognition Spoken commands belong to the most user-friendly interaction with robots. There are grammar based speech recognition systems that are optimized by understanding a subset of commands defined in a grammar. On the other hand, grammar free systems exist which are potentially able to understand all commands without prior training. In this section we present a grammar free system that integrates with the Android speech API.

Robot Face Different talking heads were developed in the last years for research in the field of human-robot interaction [4],[3],[15], [8], [12]. All of these heads were constructed in hardware, posing a challenge in designing and building these heads. A strong advantage, however, is the possibility to place cameras inside the head's eyes. This allows for intuitive interaction in a way that a person can show an object to the robot by holding it in front of the robot's head. Although this is not possible with a face completely designed in software, we chose this approach to create our animated robot face. In our opinion the high number of advantages of an animated head outweights its drawbacks. Apart from a screen, which many robots already have, there is no specific hardware that needs to be added to the robot. Moreover, it is highly customizable and can be adjusted to everyone's individual needs. Several animated robot heads were developed in the recent years that likewise possess the ability to e.g. move their lips synchronously [6], [1], [11]. In contrast to our approach, these animated heads were designed with the goal of modeling a realistic and human-like appearance. However, we focus on human robot interaction and have intentionally created an abstracted robot face with a cartoon-like appearance.

Human-like or even photo-realistic faces are employed to convey realism and authenticity to the interacting person. On the other hand the purpose of stylized cartoon faces is to invoke empathy and emotions. So far, most robots lack

[12] Example video for the robot face: `http://youtu.be/jgcztp_jAQE`

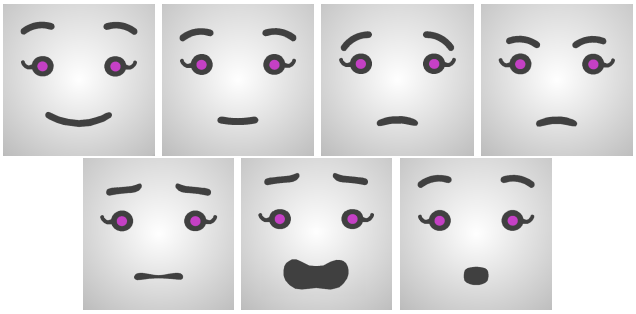


Fig. 18. Animated face of our service robot Lisa. The depicted face expressions are: happy, neutral, sad, angry, disgusted, frightened and surprised (from left to right).

humanoid features and stature, thus a realistic human face is not appropriate to interact with it. We therefore modeled an abstract cartoon face exhibiting only the most important facial features to express emotions: eyes, eyebrows and a mouth. Further, by choosing a cartoon face we minimize the risk of falling into the *uncanny valley* [9].

The lips of the robot face move synchronously to the synthesized speech. For speech synthesis the text-to-speech system *Festival* is used in the `robot_face` node. However, it can be replaced by the presented the Android speech synthesis (see below). We achieve synchronization by mapping visemes to phonemes of the synthesized text. Visemes are visually distinguishable shapes of the mouth and lips that are necessary to produce certain sounds. Phonemes are groups of similar sounds that feel alike for the speaker. Usually, only a few visemes are sufficient to achieve a realistic animation of the lips.

Additionally to the synchronized lips, the robot face is capable of expressing 6 different emotions and a neutral face. The different emotions are depicted in Fig. 18. To animate arbitrary text with the desired face expression we need dynamically generated animations. Apart from visemes we defined shape keys containing several different configurations for the eyes and eyebrows to display different face expressions. Animations are created by interpolating between the different shape keys. For animation we use Ogre3d as graphics engine, Qt as window manager and Blender for creating the meshes for the face.

6.2 ROS Environment Configuration

Speech Recognition As the communication between the Android app and the ROS node is solved by sockets you have to ensure that the ROS machine and the Android device are in the same network. An easy solution for doing this is to use your Android smartphone as wifi hotspot and connect the ROS machine to it. Additionally you have to ensure that both applications operate on the same port. You can change the port in the `recognition.launch` file in the `android_speech_pkg`. This port needs to be the same as entered in the `homer_speech` app.

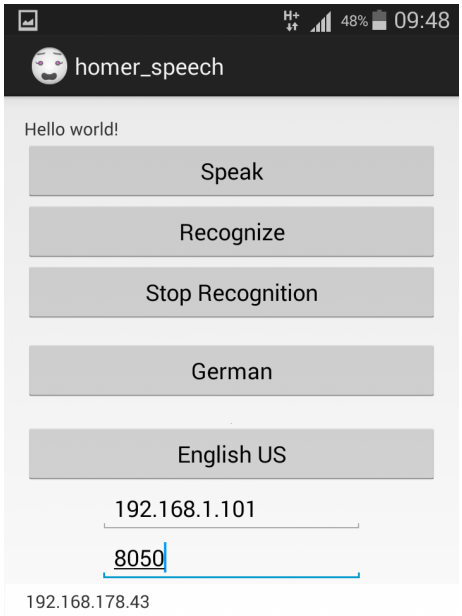


Fig. 19. The `homer_speech` app. The *Speak* and *Recognize* buttons provide speech synthesis and recognition, respectively. The two edit boxes in the lower part of the screen are used to configure the connection for speech recognition.

```
<!-- Bind to all addresses -->
<param name="~host" value="" />
<!-- Port -->
<param name="~port" value="8051" />
```

Robot Face In order to use this package you have to install the following libraries:

- Festival: `http://www.cstr.ed.ac.uk/projects/festival`
- Ogre3d: `http://www.ogre3d.org`
- Qt: `http://qt.nokia.com`
- OpenCV: `http://opencv.org` (usually comes with ROS)

If you want to create your own robot face you will additionally need Blender (`http://www.blender.org/`). You can install all of these libraries and voices for speech synthesis by opening a terminal and typing (all in one line):

```
sudo apt-get install libqt4-core libqt4-dev libqt4-gui festival
festival-dev festlex-cmu festlex-poslex festlex-oald mbrola
mbrola-us1 libogre-1.8-dev libesd0-dev libestools2.1-dev libpulse-dev
```

6.3 Quick Start and Example

Speech Recognition We provide a compiled app `homer_speech.apk` (Fig. 19) for your Android device. In order to use this app, ensure that both, the robot's computer and the smartphone are in the same network. In our tests, we attached the Android device to the robot in a comfortable height for speaking.

To use the *speech recognition*, start the app and set the robot's IP and port in the edit boxes inside the `homer_speech` app. On the robot, run the following command to start the corresponding ROS node:

```
roslaunch android_speech_pkg recognition.launch
```

The *Recognize* button initiates the recognition process and either stops listening after recognizing silence or when the *Stop Recognition* button is pressed. So far, the app allows to switch between *German* and *English US* speech recognition. The recognized speech will be send as a `std_msgs/String` on the `/recognized_speech` topic. You can test the recognition by pressing the *Recognize* button and typing the command

```
rostopic echo /recognized_speech
```

on the robot's computer.

To use the *speech synthesis*, the `host` parameter in the `synthesis.launch` file needs to be adjusted to the Android device's IP. The device's IP is shown in the `homer_speech` app in the bottom line (Fig. 19). On the robot, run the following command for speech synthesis:

```
roslaunch android_speech_pkg synthesis.launch
```

You can test your configuration either by pressing the *Speak* button or with the following command:

```
rostopic pub /speak std_msgs/String "data: 'Hello World'"
```

Robot Face After setting up the ROS environment and executing the `catkin_make` command in your ROS workspace you can start the `robot_face` by typing the following command:

```
roslaunch robot_face robot_face.launch
```

A window showing the robot face appears similar to the one shown in Fig. 20 on the left. Note that with this launch file the speech will be synthesized by *festival*. Thus, the `robot_face` can be used without any Android device. In order to use the speech synthesis of the presented app, please use the following command:

```
roslaunch robot_face robot_face_android_synthesis.launch
```

Type the following command to let the robot speak:

```
rostopic pub /robot_face/speak std_msgs/String
 "Hello, how are you?"
```

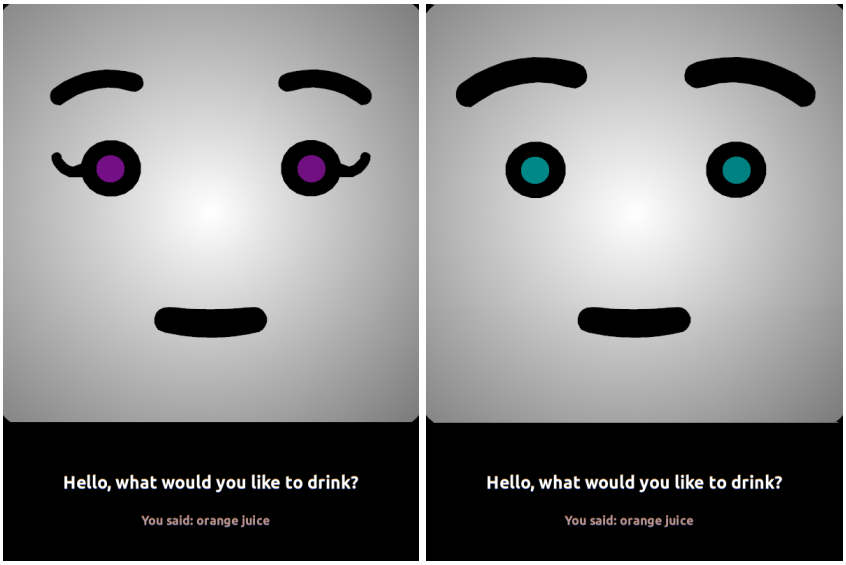


Fig. 20. The female and male version of the `robot_face` showing in white the synthesized speech and in red the recognized speech.

Table 1. Available emoticons and the corresponding face expressions.

Face expression	neutral	happy	sad	angry	surprised	frightened	disgusted
Emoticon	.	:)	:(	>:	:o	:&	:!

You will hear a voice and the face moves its lips accordingly.

If the speech recognition node is running, you can answer and see the recognized speech displayed below the face. Without speech recognition, you can fake the displayed message by typing the command:

```
rostopic pub /recognized_speech std_msgs/String
 "Thank you, I am fine!"
```

To activate different face expressions, just include the corresponding emoticon in the string that you sent to the robot. For instance, to show a happy face expression, you can type:

```
rostopic pub /robot_face/speak std_msgs/String
 "This is the happy face :)"
```

Tab. 1 shows all available face expressions and the intended emotions. Please refer to the user study in [13] on how these face expressions are perceived by humans. Apart from showing face expressions and synchronized lip movements, the `robot_face` can also show images and video streams instead of the face.

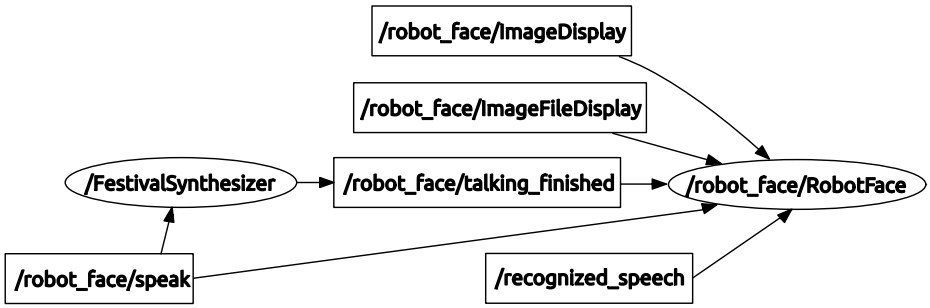


Fig. 21. Nodes and topics involved in the `robot_face` package.

Please refer to the following section to use this functionality. If you want to create your own robot face for this package, please refer to the ReadMe file[13].

6.4 Package Description and Code Examples

Speech Recognition The speech recognition component is divided into an Android app and a ROS node that converts the recognized text into a ROS conform message. The recognition is able to understand different languages that can be changed in the android system settings. An active internet connection improves the recognition results, but is not mandatory. Currently, the speech recognition needs to be started by pressing the *Recognize* button. If needed a topic for starting the recognition and sending a start recognition request can be easily integrated.

Robot Face The graph in Fig. 21 shows the two nodes contained in the `robot_face` package with all topics. The creation of phonetic features including speech synthesis is handled by the `FestivalSynthesizer` node, while the `RobotFace` node creates the corresponding animations. When using an Android device, festival still creates the phonetic features for animation. However, the Google speech API takes care of the actual speech synthesis.

The topics are explained in the following:

Publishers

`/robot_face/talking_finished` (std_msgs/String): Published once the `robot_face` has completed the speech synthesis and stopped speaking. This is used to synchronize the animation and can be used in your application to know when the speech is finished.

[13] A ReadMe is available online in the software repository.

Subscribers

`/robot_face/speak` (std_msgs/String): Text to be spoken and displayed below the face.
`/recognized_speech` (std_msgs/String): Text recognized by the speech recognition. It will be displayed below the face.
`/robot_face/ImageDisplay` (robot_face/ImageDisplay): Used to display an `std_msgs/Image` message instead of the face for a specified time.
`/robot_face/ImageFileDisplay` (robot_face/ImageFileDisplay): Used to display an image from hard disk instead of the face for a specified time.

To use all features of the `robot_face` in your application, please include the `robot_face` as *build* and *run* dependency in your package's *package.xml* file. Further, `robot_face` has do be added to the `find_package` section and as a dependency to the `catkin_package` section in the `CMakeLists.txt` file. Note, that this is only needed if you want to use the `robot_face` to display images instead of the face, since custom messages are used for this functionality. An example header to use the `robot_face` could look like this:

```
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <sensor_msgs/Image.h>
#include <robot_face/ImageDisplay.h>

class FaceExample {
  public:
    FaceExample(ros::NodeHandle nh);
    virtual ~FaceExample(){};
  private:
    void speak(std::string text);
    void talkingFinishedCallback(const std_msgs::String::ConstPtr& msg);
    void imageCallback(const sensor_msgs::Image::ConstPtr& msg);
    void send_image(std::string& path_to_image);
    ros::Publisher speak_pub_;
    ros::Publisher image_pub_;
    ros::Subscriber talking_finished_sub_;
    ros::Subscriber image_sub_;
};
```

In your class implementation you can use the following code snippets. First, include the example header and initialize all subscribers and publishers in the constructor:

```
#include "FaceExample.h"

FaceExample::FaceExample(ros::NodeHandle nh)
{
  speak_pub_ = nh.advertise<std_msgs::String>("robot_face/speak", 1);
  image_pub_ = nh.advertise<robot_face::ImageDisplay>
```

```
      ("/robot_face/ImageDisplay", 1);
  talking_finished_sub_ = nh.subscribe("robot_face/talking_finished",1,
      &FaceExample::talking_finished_callback, this);
  image_sub_ = nh.subscribe("/camera/rgb/image_color",1,
      &FaceExample::image_callback, this);
}
```

Use the following function to send speech commands to the face:

```
void FaceExample::speak(std::string text)
{
  std_msgs::String msg;
  msg.data = text;
  speak_pub_.publish(msg);
}
```

As soon as the face finishes talking, the following callback will be called:

```
void FaceExample::talkingFinishedCallback(
    const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO_STREAM("Finished speaking");
}
```

Apart from displaying text, the `robot_face` can be used to display images instead of the face. The following code snippet demonstrates how to display a `sensor_msgs/Image` message from an RGB-D camera:

```
void FaceExample::imageCallback(const sensor_msgs::Image::ConstPtr& msg)
{
  robot_face::ImageDisplay img_msg;
  img_msg.time = 0; // display time [s]
  img_msg.Image = *msg;
  image_pub_.publish(img_msg);
}
```

A display time of 0 seconds means that the image will be displayed until another message is received. To use this code, make sure that a camera is connected to your computer and the corresponding driver is loaded, e.g. by typing

```
roslaunch openni_launch openni.launch
```

To display an image from the hard disk use the following function:

```
void FaceExample::send_image(std::string& path_to_image)
{
  robot_face::ImageFileDisplay img_msg;
  img_msg.time = 0; // display time [s]
  img_msg.filename = path_to_image;
  image_pub_.publish(img_msg);
}
```

7 Conclusion

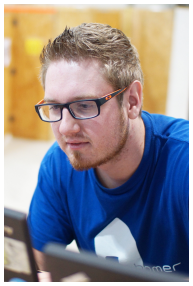
In this chapter we presented several components that are crucial for autonomous service robots and components that enhance human robot interaction. The presented mapping, navigation and object recognition are closely integrated into a graphical user interface. We believe that this close integration into a common graphical interface facilitates the development, control and monitoring of autonomous robots. Additionally, the speech recognition and animated robot face provide important interfaces for human robot interaction. We hope that this extensive tutorial together with the presented software components and videos are a valuable contribution to the ROS community.

8 Acknowledgements

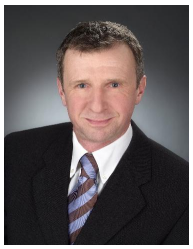
The presented software was developed in the Active Vision Group by research associates and students of the practical courses with the robots “Lisa” and “Robbie”. The authors would like to thank Dr. Johannes Pellenz, David Gossow, Susanne Thierfelder, Julian Giesen and Malte Knauf for their contributions to the software. Further, the authors would like to thank Baharak Rezvan for her assistance in testing the setup procedure of the described packages.

9 Biographies

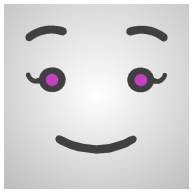
Dipl.-Inform. Viktor Seib studied Computer Sciences at the University of Koblenz-Landau where he received his Diploma in 2010. In 2011 he became a PhD student at the Active Vision Group (AGAS) at the University of Koblenz-Landau, supervised by Prof. Dr.-Ing. Dietrich Paulus. His main research areas are object recognition and service robotics. He has attended multiple RoboCup and RoCKIn competitions and is currently team leader and scientific superviser of the RoboCup@Home team homer@UniKoblenz.



B.Sc. Raphael Memmesheimer finished his Bachelor in Computational Visualistics and is currently in his master studies focusing on robotics and computer vision. His special interests are visual odometry, visual SLAM and domestic service robots. As research assistant he is working in the Active Vision Group (AGAS) and has successfully attended multiple RoboCup and RoCKIn competitions. Currently he is technical chief designer of the homer@UniKoblenz RoboCup@Home team and states that ”Robots are the real art”!



Prof. Dr.-Ing. Dietrich Paulus obtained a Bachelor degree in Computer Science from University of Western Ontario, London, Ontario, Canada, followed by a diploma (Dipl.-Inform.) in Computer Science and a PhD (Dr.-Ing.) from Friedrich-Alexander University Erlangen-Nuremberg, Germany. He worked as a senior researcher at the chair for pattern recognition (Prof. Dr. H. Niemann) at Erlangen University from 1991-2002. He obtained his habilitation in Erlangen in 2001. Since 2001 he is at the institute for computational visualistics at the University Koblenz-Landau, Germany where he became a full professor in 2002. From 2004-2008 he was the dean of the department of computer science at the University Koblenz-Landau. Since 2012 he is head of the computing center in Koblenz. His primary research interest are active computer vision, object recognition, color image processing, medical image processing, vision-based autonomous systems, and software engineering for computer vision. He has published over 150 articles on these topics and he is the author of three textbooks. He is member of Gesellschaft für Informatik (GI) and IEEE.



Lisa is the robot of team homer@UniKoblenz. She first participated in the RoboCup GermanOpen and the RoboCup World Championship in 2009. Since then, her design was continuously changed and improved to better adapt to the requirements of service robot tasks. Lisa is developed by PhD, master and bachelor students in practical courses of the Active Vision Group. Lisa and team homer@UniKoblenz won the 1st Place in the RoboCup@Home World Championship 2015 in Hefei, China.

References

1. I. Albrecht, J. Haber, K. Kahler, M. Schroder, and H.P. Seidel. May i talk to you?:-)-facial animation from text. In *Computer Graphics and Applications, 2002. Proceedings. 10th Pacific Conference on*, pages 77–86. IEEE, 2002.
2. Herbert Bay, Tinne Tuytelaars, and Luc J. Van Gool. SURF: Speeded up robust features. *ECCV*, pages 404–417, 2006.
3. C. Breazeal. Toward sociable robots. *Robotics and Autonomous Systems*, 42(3):167–175, 2003.
4. C. Breazeal and B. Scassellati. How to build robots that make friends and influence people. In *Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on*, volume 2, pages 858–863. IEEE, 1999.
5. W. Eric L. Grimson and Daniel P. Huttenlocher. On the sensitivity of the hough transform for object recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-12(3):255–274, 1990.

6. Joakim Gustafson, Magnus Lundeberg, and Johan Liljencrants. Experiences from the development of august - a multi- modal spoken dialogue system. In *ESCA Workshop on Interactive Dialogue in Multi- Modal Systems (IDS-99)*, pages 61–64, 1999.
7. David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
8. Ingo Lütkebohle, Frank Hegel, Simon Schulz, Matthias Hackel, Britta Wrede, Sven Wachsmuth, and Gerhard Sagerer. The bielefeld anthropomorphic robot head flobi. In *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, 5 2010. IEEE, IEEE.
9. Masahiro Mori, Karl F MacDorman, and Norri Kageki. The uncanny valley [from the field]. *Robotics & Automation Magazine, IEEE*, 19(2):98–100, 2012.
10. Marius Muja. Flann, fast library for approximate nearest neighbors, 2009. http://mloss.org/software/view/143/.
11. A. Niswar, E.P. Ong, H.T. Nguyen, and Z. Huang. Real-time 3d talking head from a synthetic viseme dataset. In *Proc. of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 29–33. ACM, 2009.
12. Javier Ruiz-del-Solar, Mauricio Mascaró, Mauricio Correa, Fernando Bernuy, Romina Riquelme, and Rodrigo Verschae. Analyzing the human-robot interaction abilities of a general-purpose social robot in different naturalistic environments. In *Lecture Notes in Computer Science (RoboCup Symposium 2009)*, volume 5949 of *LNCS*, pages 308–319, 2009.
13. Viktor Seib, Julian Giesen, Dominik Grüntjens, and Dietrich Paulus. Enhancing human-robot interaction by a robot face with facial expressions and synchronized lip movements. In Vaclav Skala, editor, *21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2013.
14. Viktor Seib, Michael Kusenbach, Susanne Thierfelder, and Dietrich Paulus. Object recognition using hough-transform clustering of surf features. In *Workshops on Electronical and Computer Engineering Subfields*, pages 169 – 176. Scientific Cooperations Publications, 2014.
15. S. Sosnowski, A. Bittermann, K. Kuhnlenz, and M. Buss. Design and evaluation of emotion-display eddie. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3113–3118. IEEE, 2006.
16. Stephan Wirth and Johannes Pellenz. Exploration transform: A stable exploring algorithm for robots in rescue environments. In *Safety, Security and Rescue Robotics, 2007. SSRR 2007. IEEE International Workshop on*, pages 1–5, 2007.
17. Alexander Zelinsky. Robot navigation with learning. *Australian Computer Journal*, 20(2):85–93, 5 1988.
18. Alexander Zelinsky. *Environment Exploration and Path Planning Algorithms for a Mobile Robot using Sonar*. PhD thesis, Wollongong University, Australia, 1991.